BubbleSched : construire son propre ordonnanceur de threads pour machines multiprocesseurs hirarchiques

Samuel Thibault

RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

BubbleSched : construire son propre ordonnanceur de threads pour machines multiprocesseurs hiérarchiques

Samuel Thibault — samuel.thibault@labri.fr

Projet INRIA RUNTIME
LABRI — Université Bordeaux 1 — 351 cours de la libération — 33405 Talence cedex

Résumé
L'efficacité de l'exécution d'une application multithreadée irrégulière sur une architecture multiprocesseurs fortement hiérarchique repose essentiellement sur la qualité de l'ordonnancement des threads et du placement des données. Pour obtenir d'excellentes performances, les programmeurs sont souvent contraints de sacrifier la portabilité de leur application en câblant dans celle-ci des stratégies de placement *ad-hoc* fortement dépendantes de l'architecture. Pour remédier à ce problème de portabilité des performances, nous avons défini une plate-forme permettant de décrire dynamiquement la structure hiérarchique des calculs et de définir simplement des ordonnanceurs dédiés, efficaces et portables. Nous justifions l'intérêt d'une telle approche et décrivons la technique que nous avons mise au point pour définir simplement de tels ordonnanceurs.

Mots-clés : Threads, Scheduling, Bubbles, NUMA, SMP.

1. Introduction

Après s'être livrés durant 30 ans à une course à la fréquence, les constructeurs de microprocesseurs rivalisent maintenant à coups de puces multi-core SMT. Ainsi, SUN commercialise des puces capables d'exécuter simultanément 32 threads sur 8 cores et IBM en est à 4 threads sur 2 cores. De leur côté AMD et INTEL ont planifié la production de puces 4-cores et réfléchissent même à l'intégration d'une centaine de cores sur une seule puce. De plus, les serveurs de calculs sont souvent à mémoire partagée et organisés hiérarchiquement (SUN WILDFIRE, SGI ALTIX, ou encore IBM P560Q).
Exploiter efficacement de telles machines hiérarchiques est très complexe. En particulier, l'ordonnanceur de threads est face à quelques dilemmes. Ainsi, comme sur de telles architectures le temps d'accès à la mémoire dépend de la position relative du processeur et du banc mémoire (facteur NUMA), l'ordonnanceur doit faire en sorte qu'un thread soit exécuté à proximité des données qu'il manipule. D'un autre côté, les cores d'une même puce se partageant la bande passante mémoire, l'ordonnanceur doit distribuer les threads les plus gourmands en bande passante sur des puces différentes.
Ainsi, ordonnancer une application multithreadée en tenant compte de la hiérarchie mémoire, de celle des unités de calcul et des impératifs de l'application est un vrai défi. Certes les systèmes d'exploitation munis d'ordonnanceurs généralistes peuvent, dans une certaine mesure, prendre en compte l'état de la machine (via des mesures de performance) mais ils ne tiennent pas compte de l'état de l'application exécutée. Or ces dernières peuvent avoir un comportement très irrégulier, imprédictible avant leur exécution (cas du maillage adaptatif) et qui peuvent provoquer d'importants problèmes d'équilibrage de charge. Du point de vue des performances, la solution idéale serait que l'application décide elle-même de l'ordonnancement de ses threads et donc d'utiliser un ordonnanceur dédié à l'application. Cependant écrire un ordonnanceur, qui plus est portable, est techniquement très difficile.
Pour remédier à ce problème de portabilité des performances, nous avons réalisé une plate-forme permettant aux programmeurs (de bibliothèques scientifiques spécialisées ou d'environnements de programmation parallèle) d'écrire simplement des ordonnanceurs exécutant efficacement leur applications sur les machines multiprocesseurs. Pour ce faire, comme nous l'avons décrit dans [1], l'application décrit dynamiquement la structure hiérarchique de ses calculs en regroupant dans des *bulles* imbriquées les

threads qui partagent des intérêts communs. Une API de haut niveau permet de manipuler ces bulles et de construire des ordonnanceurs performants et portables sans pour autant se plonger dans de sordides détails techniques.
Dans cet article, nous rappelons les points clés nécessaires à la conception d'un ordonnanceur de threads spécialisé, puis présentons notre plate-forme et illustrons la programmation d'ordonnanceurs, nous abordons ensuite quelques éléments d'implémentation. Nous comparons ensuite notre solution aux travaux apparentés avant de conclure.

2. Développement d'ordonnanceurs spécialisés

Ordonnancer des threads signifie trouver un compromis entre de nombreuses contraintes : favoriser les affinités entre threads, cache et mémoire, profiter de toute la puissance processeur, réduire les coûts de synchronisation et respecter les contraintes applicatives.

2.1. Indications utiles à l'ordonnancement

L'ordonnanceur dispose de quantité de contraintes, indications, constantes et métriques.
À l'exécution, il dispose d'informations sur la structure de la machine cible, depuis l'organisation des bancs mémoire pour les machines NUMA jusqu'à l'organisation des caches sur les puces multicores. Il est également possible de savoir si l'application s'exécute correctement à l'aide de *compteurs de performances*. Le nombre d'accès mémoire distants permet d'effectuer par exemple des migrations de pages mémoire [2]. Le nombre de défauts de cache et d'instructions par cycle (IPC) permettent également de mesurer la concurrence au sein des processeurs.
Le *compilateur* peut aussi fournir des indications sur le comportement de l'application. En analysant les schémas d'accès aux données [3], on peut estimer comment répartir threads et données pour favoriser les accès locaux et s'assurer que l'occupation mémoire sera possible sur la machine cible.
Enfin, les *programmeurs* eux-mêmes détiennent *beaucoup* d'informations. Ils savent si le schéma d'accès aux données est plutôt orienté producteur/consommateur ou bien complètement irrégulier par exemple. L'ordonnanceur peut alors coopérer avec le gestionnaire de mémoire pour établir des stratégies telles que « allocations entrelacées + ordonnancement centralisé », ou bien « allocations localisées + ordonnancement hiérarchisé ». Les programmeurs ont aussi une assez bonne idée du comportement des threads vis-à-vis de l'ordonnancement : s'ils s'endorment souvent ou bien consomment beaucoup de temps processeur par exemple, ce qui permet de trouver de bonnes combinaisons de politiques d'ordonnancement. Parfois, les schémas de synchronisation sont connus : on peut savoir que la libération d'un sémaphore devrait ordonnancer tout de suite le thread réveillé. Un thread endormi à court terme sur un mutex ne devrait par ailleurs pas être compté comme vraiment inactif pour l'équilibrage de charge.

2.2. Limites des ordonnanceurs classiques

Comme nous l'avons décrit dans [1], différentes approches d'ordonnancement sont actuellement utilisées. L'approche *opportuniste*, approche gloutonne plus ou moins distribuée, ne prend en général pas en compte les informations d'affinités que peuvent fournir le compilateur ou les programmeurs. L'approche *prédéterminée*, où un ordonnancement statique est pré-établi à l'aide de modélisations du calcul et de la machine cibles, ne peut pas, par nature même, s'appliquer aux problèmes très irréguliers où le comportement du calcul change selon les résultats intermédiaires même. Enfin, les approches *négociées* basées sur des extensions de langage (OpenMP, HPF) fournissent aux programmeurs une expressivité insuffisante pour exprimer du parallélisme irrégulier.
Les applications et leur comportement sont si divers qu'un ordonnanceur universel semble ne pas pouvoir être efficace. Une solution serait donc de demander aux programmeurs d'écrire des ordonnanceurs dédiés, à l'écoute de l'application. Effectuer ceci dans le noyau est assez irréaliste, que ce soit en termes de développement ou de déploiement. Cependant, comme la plupart des systèmes d'exploitation permettent de fixer des threads sur les processeurs, il est concevable de réaliser un ordonnancement de niveau utilisateur, mais cela nécessite tout de même une expertise certaine.

2.3. Savoir-faire technique et algorithmique commun à tout ordonnanceur

Tout d'abord, il faut avoir une idée des articulations nécessaires entre les fonctions de l'ordonnanceur. Par exemple, lorsqu'un thread T essaie de prendre un mutex occupé, il doit s'endormir par un appel à la

fonction `sleep()` qui doit appeler une fonction `find_next()` pour véritablement suspendre le thread T en en trouvant un autre à exécuter. La fonction `sleep()` doit cependant faire attention à la possibilité d'un appel concurrent à la fonction `wake_up()`, effectué sur un autre processeur par un autre thread qui est précisément en train de libérer le mutex que T essayait de prendre ! De tels problèmes, courants dans les ordonnanceurs, induisent des bugs difficiles à cerner.
Écrire un ordonnanceur *portable* est également difficile : pour faire fonctionner un ordonnanceur de threads, il faut de nombreuses connaissances sur le système d'exploitation et le processeur cibles. Par exemple, préempter un thread actif pour basculer l'exécution sur un autre est une opération dont l'implémentation dépend du système d'exploitation et du processeur. Pour respecter les conventions binaires (ABI) du système, il faut correctement sauver et restaurer l'état des threads : non seulement les registres du processeur, mais aussi les objets spéciaux tels que la variable `errno`.
La qualité de l'implémentation est importante pour obtenir un ordonnanceur *efficace* et *passant à l'échelle* : comme le détaille EDLER [4], toutes les briques de bases doivent être soigneusement implémentées. Les verrous rotatifs, par exemple, devraient non seulement être écrits en assembleur (ce qui implique des problèmes de portage), mais doivent aussi être distribués afin d'éviter des contentions mémoire [5].

2.4. Discussion

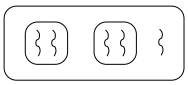
Le problème est donc de tirer parti des informations disponibles afin d'établir un ordonnancement qui est fatalement un compromis puisque, par exemple, distribuer les threads sur toute la machine permet de profiter des processeurs mais concentrer les threads autour des données traitées permet de gagner en affinités. Certaines parties d'application peuvent même réclamer des politiques d'ordonnancement différentes. Les approches actuelles ne sont donc pas à même de prendre en compte ces contraintes, et de manière plus générale, manquent d'interaction avec les programmeurs et le compilateur.
Une direction possible est de fournir les outils pour renforcer la collaboration entre l'application, l'ordonnanceur et la machine : des outils pour structurer, qualifier le parallélisme de l'application et des outils pour piloter son ordonnancement. Pour des raisons de portabilité, ces outils devraient évidemment s'appuyer sur des abstractions de haut niveau de la machine sous-jacente.

3. BubbleSched : une boîte à outil puissante et portable pour ordonnancer des threads

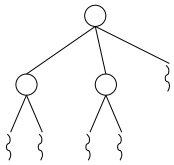
Nous avons développé une plate-forme facilitant l'écriture d'ordonnanceurs efficaces, flexibles et portables sur les machines hiérarchiques. Cette plate-forme est basée sur des abstractions de haut niveau appelées *Bulles*, permettant la description dynamique de la structure parallèle des applications.

3.1. Des bulles et des listes de tâches pour modéliser l'application et la machine

Dans [1], nous avons proposé de modéliser les relations entre les threads d'une application à l'aide d'ensembles récursifs appelés **bulles**. La figure 1 montre un exemple d'une telle modélisation : quatre threads de calcul sont groupés par paires (parce qu'ils travaillent sur les mêmes données) dans des bulles, qui sont elles-mêmes regroupées avec un thread de communication au sein d'une plus grande bulle. Ceci peut ainsi exprimer des relations telles que le partage de données, la participation à des opérations collectives, un bon comportement lors d'un co-ordonnancement sur processeur SMT, ou plus généralement un besoin de politique d'ordonnancement particulier (sérialisation, préemption, gang scheduling, etc.).
Les bulles ont également un ensemble d'attributs qui peuvent être spécifiés par les programmeurs pour donner des informations sur l'ensemble des threads contenus : priorité, « élasticité », estimation de l'utilisation processeur ou mémoire, politique d'ordonnancement, etc. Des compteurs peuvent également permettre de connaître le nombre total de threads, de threads actifs, ou la quantité de mémoire allouée.
De manière similaire à la hiérarchie de listes de Nano-Threads de NIKOLOPOULOS *et al.* [6], nous modélisons les machines hiérarchiques à l'aide d'une hiérarchie de listes de threads. Chaque élément de chaque niveau de la machine est représenté par une liste de threads. La figure 2 montre comment une machine hiérarchique complexe est modélisée. La machine en entier, chaque nœud NUMA, chaque puce, chaque core et chaque processeur logique SMT est représenté par une liste de threads.
Notre politique d'ordonnancement de base est alors un algorithme de Self-Scheduling hiérarchique : lorsqu'il est inactif, un processeur parcourt de bas en haut toutes les listes de threads qui le couvrent, et



(a) Représentation graphique.



(b) Représentation arborescente.

```
marcel_t cpu_thread[4], comm_thread;
marcel_bubble_t bubbles[2], bubble;

marcel_bubble_insertthread(&bubbles[0],&thread[0]);
marcel_bubble_insertthread(&bubbles[0],&thread[1]);
marcel_bubble_insertthread(&bubbles[1],&thread[2]);
marcel_bubble_insertthread(&bubbles[1],&thread[3]);

marcel_bubble_insertbubble(&bubble,&bubbles[0]);
marcel_bubble_insertbubble(&bubble,&bubbles[1]);
marcel_bubble_insertthread(&bubble,&comm_thread);
```

(c) Code source correspondant.

FIG. 1 – Expression des relations entre threads.

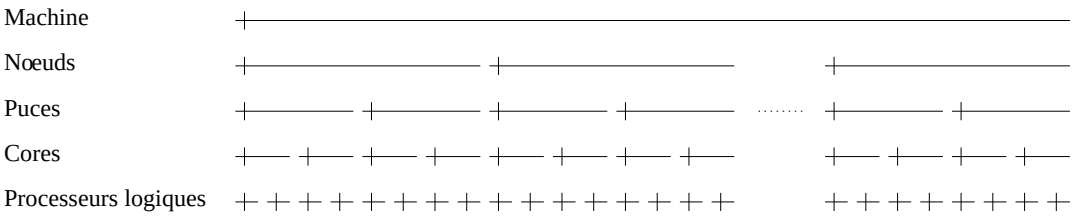
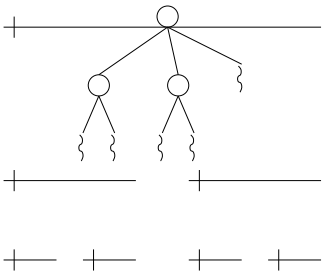
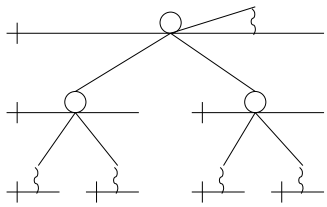


FIG. 2 – Modélisation d'un machine très hiérarchique.



(a) Bonne utilisation processeur, affinités non prises en compte.



(b) Utilisation distribuée des processeurs, affinités correctement prises en compte.

FIG. 3 – Distributions possibles des threads et bulles sur la machine.

exécute le premier thread qu'il trouve. La liste de threads sur laquelle un thread est placé exprime ainsi son domaine d'ordonnancement : si le thread est placé sur une liste représentant une puce physique, il sera ordonnancé par les processeurs de cette puce ; s'il est placé sur la liste de la machine entière, il pourra être ordonnancé par n'importe quel processeur de la machine. De plus, les priorités sont prises en compte : si un thread de forte priorité se réveille sur la liste de la machine entière, un processeur l'exécutera dès que possible, même si des threads plus « locaux » sont disponibles. Cela permet par exemple aux threads de communication d'être exécutés dès que possible.

3.2. Des outils pour implémenter des stratégies d'ordonnancement

Par défaut, les bulles (et donc les threads contenus dedans) sont placées au démarrage sur la liste de la machine entière. Lorsque l'on exécute l'exemple précédent de la figure 1 sur une machine double bi-core, on obtient la figure 3(a) : la hiérarchie de bulles reste en haut de la hiérarchie de listes. Une telle distribution permet certes d'utiliser tous les processeurs de la machine, puisque tous ont l'opportunité d'exécuter n'importe quel thread. Cela ne prend cependant pas en compte les affinités entre threads et processeurs, puisqu'aucune relation entre eux n'est utilisée. À l'inverse, la figure 3(b) montre comment les threads peuvent être distribués en prenant fortement les affinités en compte, puisque les threads sont en fait correctement répartis sur les processeurs selon leurs affinités. Cependant, si certains threads s'endorment, les processeurs correspondant deviennent inactifs, et l'utilisation des processeurs est donc potentiellement partielle.

Nous avons développé une interface de programmation pour répartir facilement bulles et threads sur les listes afin d'obtenir des distributions telles que celle de la figure 3(b). Threads et bulles sont indif-

```
/* Niveaux de listes. */
runqueue_t main_rq, node_rq[], chip_rq[],
  core_rq[], cpu_rq[];
/* Verrouillage. */
void runqueue_lock(runqueue_t *rq);
void runqueue_unlock(runqueue_t *rq);
void all_lock(); all_unlock();
```

```
/* Opérations sur listes. */
int runqueue_empty(runqueue_t *rq);
entity_t *runqueue_entry(runqueue_t *rq);
runqueue_foreach(runqueue_t *rq, entity_t **e)
void deactivate(entity_t *e, runqueue_t *rq);
void activate(entity_t *e, runqueue_t *rq);
```

FIG. 4 – Interface de programmation

féremment considérés comme **entités**, une bulle contenant simplement des entités (threads ou autres bulles). Des primitives sont alors fournies pour manipuler les entités sur les listes de la figure 3(b) (appelées `runqueues`), voir figure 4. On peut accéder aux listes à l'aide de simples tableaux (des pointeurs père/fils sont également disponibles pour des parcours arborescents). Pour gérer la concurrence, il est possible de verrouiller finement les listes, mais les programmeurs pourront parfois préférer se contenter d'un verrouillage total pendant leurs manipulations (`all_lock`). On peut alors énumérer les entités d'une liste, et décider d'en enlever (`deactivate`) ou en ajouter (`activate`).
Écrire un ordonnanceur de haut niveau se réduit alors à écrire quelques fonctions clés. La fonction `ma_bubble_schedule()` est appelée lorsque l'ordonnanceur de Self-Scheduling de base rencontre une bulle lors de sa recherche du thread suivant à exécuter. L'implémentation par défaut recherche simplement un thread dans la bulle (ou l'une de ses sous-bulles), et l'exécute. La fonction `ma_bubble_tick()` est appelée lorsque le quantum de temps d'une bulle expire, et permet ainsi des opérations périodiques sur les bulles avec une notion de temps relative à la bulle. Bien sûr, des threads peuvent aussi être utilisés pour effectuer des opérations en arrière-plan. Au final, les programmeurs peuvent manipuler la répartition des threads avec un haut niveau d'abstraction en décidant du placement des entités sur les listes.

3.3. Exemples d'implémentations d'algorithmes d'ordonnancement

Notre interface de programmation permet de développer des « ordonnanceurs à bulles » très variés.

3.3.1. Ordonnancement à éclatement

Un premier exemple d'ordonnanceur possible est l'« algorithme à éclatement » décrit dans un précédent papier [1] : un algorithme de Self-Scheduling « tire » vers les processeurs les bulles, qui « éclatent » (*i.e.* déversent leur contenu sur la liste), d'une manière opportuniste, lorsqu'elles ont atteint un niveau hiérarchique donné par les programmeurs. Pour implémenter cela, la fonction `ma_bubble_schedule()` tire légèrement la bulle vers le processeur, et le contenu est déversé le cas échéant. En quelques itérations, bulles et threads sont répartis comme sur la figure 3(b). De plus, les bulles sont automatiquement régénérées à l'aide de la fonction `ma_bubble_tick()` : leur contenu originel y est replacé, et elles sont remises sur la liste de la machine, pour être de nouveau distribuées. Cela permet d'adapter automatiquement la distribution à une nouvelle charge de calcul, tout en prenant en compte les affinités.

3.3.2. Ordonnancement de gangs

Avec l'émergence dans les années 80 des réseaux de machines multiprocesseurs, OUSTERHOUT [7] propose de regrouper les threads et données par affinités sous forme de *gangs* et d'ordonnancer non plus des threads sur des processeurs mais des gangs sur des machines. Dans notre environnement, pour réaliser un *gang-scheduler*, on associe une bulle à chaque gang, et l'on laisse tourner un démon écrit en une simple douzaine de lignes, voir figure 5(a). Il utilise une liste `nosched_rq` supplémentaire, non consultée par l'ordonnanceur de base, où il met de côté toutes les bulles (i.e. les gangs) sauf une, qu'il laisse sur la liste principale pendant un certain laps de temps, avant de recommencer pour placer une autre bulle. Il nous semble qu'un programmeur peut modifier un tel algorithme sans connaissances techniques.
Une utilisation intéressante de ce gang scheduler est l'émulation équitable d'un réseau de machines virtuelles sur une même machine physique. Chaque machine virtuelle est représentée par une bulle. Puisque le gang scheduler, en arrière-plan, donne tour à tour à chaque bulle une tranche de temps, il ordonnance ainsi équitablement les machines virtuelles. Nous avons réalisé une expérience où trois gangs de calcul intensif se partagent une machine à quatre processeurs. On observe sur la figure 5(b) que la machine est bien partagée de manière équitable : le gang 0 formé de 5 threads (notés `0-[0-4]`)

```
runqueue_t nosched_rq;
while(1) {
  runqueue_lock(&main_rq);
  runqueue_lock(&nosched_rq);
  /* Remettre toute bulle de côté. */
  runqueue_for_each_entry(&main_rq, &e) {
    deactivate_entity(e, &main_rq);
    activate_entity(e, &nosched_rq);
  }
  /* Placer la bulle suivante
  sur la liste principale. */
  if (!runqueue_empty(&nosched_rq)) {
    e = runqueue_entry(&nosched_rq);
    deactivate_entity(e, &nosched_rq);
    activate_entity(e, &main_rq);
  }
  runqueue_unlock(&main_rq);
  runqueue_unlock(&nosched_rq);
  /* Laisser cette bulle s'exécuter un peu. */
  marcel_delay(timeslice);
}
```

(a) Code source.

nom	pr	% proc	s	proc
gang scheduler	42	0.0	I	0
0-0	43	26.4	R	3
0-1	43	26.6	R	0
0-2	43	26.8	R	2
0-3	43	25.6	R	1
0-4	43	26.6	R	3
1-0	43	21.9	R	2
1-1	43	22.2	R	0
1-2	43	22.5	R	3
1-3	43	22.1	R	2
1-4	43	21.9	R	0
1-5	43	22.6	R	1
2-0	43	19.2	R	3
2-1	43	18.9	R	1
2-2	43	19.3	R	0
2-3	43	19.8	R	2
2-4	43	19.1	R	3
2-5	43	19.3	R	1
2-6	43	19.2	R	2

(b) Statistiques d'exécution (outil *a la* `top`)

FIG. 5 – Un gang scheduler de base.

obtient, tout comme les autres gangs, à peu près 133% de temps processeur (sur 400%).
Dans ce modèle originel du gang scheduling, des processeurs peuvent rester inactifs parce qu'une même machine ne peut exécuter qu'un gang à la fois, même si celui-ci est « petit ». FEITELSON *et al.* [8] proposent un contrôle hiérarchique des processeurs, pour pouvoir exécuter plusieurs gangs sur la même machine. Une telle approche peut être facilement implémentée en exécutant un thread de gang scheduling pour chaque nœud de la hiérarchie de la machine. Une certaine synchronisation est nécessaire entre ces threads, mais cela peut être facilement effectué à l'aide de sémaphores usuels.

3.3.3. Ordonnancement par vol de travail

Un de nos algorithmes en cours de développement est basé sur le vol de travail : la hiérarchie de bulle est d'abord placée sur la liste du processeur 0. Lorsque la fonction `ma_bubble_schedule()` est appelée par un processeur inactif, elle utilise les pointeurs père/fils des listes pour chercher du travail à voler localement d'abord (sur la liste de threads de l'autre processeur du même core par exemple), puis plus globalement, jusqu'à trouver du travail. Savoir quel « vol » effectuer est un problème algorithmique non trivial : seule une partie de la hiérarchie de bulles devrait être tirée vers le processeur inactif. De plus, la structure de la hiérarchie devrait être prise en compte : il s'agit donc véritablement d'une sorte d'« étirement local de l'arbre des bulles » le long de la machine. Tous les attributs et statistiques attachées aux bulles décrits dans la section précédente devraient également être soigneusement pris en compte de manière heuristique pour obtenir une répartition bien adaptée à l'application. En effet, selon les applications il faudra privilégier plutôt une répartition de l'occupation mémoire, ou parfois plutôt de la bande passante nécessaire ; un compromis est à trouver. Ces problèmes ne sont cependant que purement algorithmiques : il n'y a plus de problème technique.

3.3.4. Ordonnancements mixtes

Les programmeurs peuvent même combiner tous ces ordonnanceurs. Une partie de la machine peut exécuter une partie de l'application avec certaines politiques, pendant que d'autres parties de la machine exécutent d'autres parties de l'application avec d'autres politiques, obtenant ainsi une combinaison *spatiale*. En étendant l'implémentation du gang scheduling décrite ci-dessus, chaque gang peut avoir sa propre politique d'ordonnancement. Ou bien l'application peut demander différents ordonnancements selon ses phases de calculs. On obtient ainsi une combinaison *temporelle* d'ordonnanceurs. Des priorités fortes définies par le programmeur peuvent également fournir des *niveaux* d'ordonnancement : une partie L de faible priorité de l'application peut être préemptée par une partie H de priorité plus forte de l'application, qui peut alors être ordonnancée avec une autre politique ; mais si une partie de la machine n'est pas utilisée par la partie H, la partie L peut continuer à être ordonnancée sur les processeurs restants, avec sa politique habituelle.

	Yield (ns)	Switch (ns)	Total
MARCEL (originel)	240	270	510
MARCEL (bulle)	247	284	531
NPTL (Linux 2.6.15)	820	410	1230

TAB. 1 – Coût de l'ordonnanceur modifié de MARCEL. Yield : parcours des listes seulement ; Switch : synchronisation et changement de contexte.

4. Implémentation

MARCEL [9, 10] est une bibliothèque de threads à deux niveaux : elle fixe un thread de niveau noyau sur chaque processeur, et effectue ensuite des changements de contexte entre des thread utilisateurs. Ainsi, en supposant qu'aucune autre application ne tourne sur la machine, MARCEL garde un contrôle complet sur l'ordonnancement des threads sur les processeurs. Notre plate-forme BubbleSched est implémentée au sein de l'ordonnanceur de MARCEL en ajoutant les appels aux fonctions cités dans la section 3.2.
Pour faciliter la programmation, de la même façon que threads et bulles sont indifféremment considérés comme entités, listes et bulles sont considérés comme *conteneurs* : ils contiennent une liste d'entités.
La gestion du placement des bulles et des threads est un problème ardu. Migrer des threads peut être particulièrement délicat : il peut s'agir de placer sur une liste un thread contenu dans une bulle, alors qu'il est encore en train de s'exécuter sur une autre liste. Il est ainsi nécessaire de mémoriser ces trois informations pour pouvoir gérer tous les cas : le conteneur *initial* d'une entité est celui que les programmeurs ont défini initialement, et doit être préservé puisqu'il représente l'affinité entre les entités (les programmeurs peuvent cependant le changer si ces affinités elles-mêmes évoluent). Le conteneur d'*ordonnancement* d'une entité est le conteneur où l'entité devrait être ordonnancée. Elle n'est pas forcément déjà en train de s'y exécuter, mais elle devrait y parvenir dans un délai bref. Le conteneur d'*exécution* est enfin le conteneur où une entité s'exécute et pour lequel elle contribue aux statistiques.
La gestion du verrouillage est particulièrement importante : pour des raisons de performances, il devrait être le plus distribué possible. Cependant, distribuer les verrous impose de faire très attention aux interblocages. Notre convention pour verrouiller plusieurs conteneurs est de le faire de haut en bas : par exemple, la liste de la machine d'abord, puis les listes des nœuds, puis les bulles placées sur ces listes, puis les bulles contenues dans ces bulles, etc. Dans certaines situations, il est alors nécessaire de libérer un verrou pour pouvoir en prendre un autre. De telles situations se sont cependant révélées rares.
Enfin notons que l'introduction de l'API de programmation des ordonnanceurs n'a pas eu d'impact négatif sur les perfomances. En effet, nous avons mesuré l'impact de notre implémentation sur un PENTIUM IV XEON à 2,66 GHz, voir table 1. On observe pour la recherche du prochain thread à exécuter (temps *Yield*) et pour le changement de contexte proprement dit (temps *Switch*) un léger surcoût, mais les résultats restent très bons compartivement à ceux de NPTL (LINUX 2.6).

5. Travaux apparentés

Bossa [11] fournit des abstractions d'ordonnancement de haut niveau et un langage pour développer et prouver des ordonnanceurs. L'implémentation actuelle est basée sur le noyau LINUX 2.2. Cependant, l'objectif ici est surtout de pouvoir *prouver* la correction d'un ordonnanceur. Par conséquent, le langage proposé, bien que suffisamment puissant pour implémenter l'ordonnanceur mono-processeur de LINUX 2.2, est assez restrictif, et limite ainsi beaucoup les programmeurs.
Le projet ELITE [12] fournit une très bonne implémentation d'une plate-forme d'ordonnancement de niveau utilisateur qui prend en compte les affinités entre threads, cache et données. Des modèles mathématiques sont même utilisés pour calculer les probabilités des fautes de cache. Cependant, l'interaction avec l'application est très faible : les affinités sont détectées plutôt que fournies par les programmeurs.
Plusieurs systèmes d'exploitation fournissent des outils pour distribuer des threads sur la machine en les rassemblant dans des ensembles : liblgroup sur Solaris [13] et NSG sur Tru64 [14]. De tels ensembles ressemblent beaucoup à des bulles sans notion de récursion. Par nature même, aucun de ces outils n'est vraiment portable, mais surtout, aucun d'entre eux ne fournit le degré de contrôle que nous fournissons : avec la plate-forme BubbleSched, l'application peut fournir des fonctions qui s'insèrent au cœur même de l'ordonnanceur, pour pouvoir réagir à des évènements tels que le *réveil* de thread ou l'*inactivité* des

processeurs. Un tel degré d'interaction avec l'ordonnanceur noyau est impossible.

6. Conclusion

Dans ce papier, nous avons présenté la plate-forme BubbleSched. Elle fournit aux programmeurs un moyen d'exprimer la structure parallèle des applications ainsi que des primitives d'ordonnancement de haut niveau. Au final, les programmeurs peuvent « piloter » finement l'ordonnanceur en écrivant quelques fonctions s'insérant au sein l'ordonnanceur. Des exemples d'implémentation de stratégies d'ordonnancement ont montré combien cela était à la fois facile et puissant. Bien sûr, écrire de telles fonctions est difficile du point de vue algorithmique, mais les programmeurs peuvent vraiment laisser les détails techniques de côté pour se concentrer sur les problèmes algorithmiques.

Ces travaux ouvrent de nombreuses perspectives. À court terme, un mécanisme générique pour les attributs et statistiques sur les bulles sera développé comme aide à la décision : les programmeurs d'application pourraient même fournir leurs propres outils de mesure, que la plate-forme BubbleSched synthétiserait selon la hiérarchie de bulles. Plusieurs approches algorithmiques pourront alors être implémentées, testées et affinées pour ordonnancer de véritables applications.

À plus long terme, un ordonnanceur générique pourrait être développé ; celui-ci prendrait en compte autant d'informations qu'il est possible de collecter auprès du matériel, du compilateur et du programmeur. L'intégration au sein du noyau LINUX pourrait même être envisagée, puisqu'une hiérarchie naturelle de bulles existe déjà au travers des notions de threads, processus, jobs, sessions et utilisateurs.

Bibliographie

1. THIBAULT (S.), « A flexible thread scheduler for hierarchical multiprocessor machines », dans *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge / USA, 06 2005. ICS / ACM / IRISA.
2. MARATHE (J.) et MUELLER (F.), « Hardware profile-guided automatic page placement for ccnuma systems », dans *Sixth Symposium on Principles and Practice of Parallel Programming*, mars 2006.
3. SHEN (X.), GAO (Y.), DING (C.) et ARCHAMBAULT (R.), « Lightweight reference affinity analysis », dans *19th ACM International Conference on Supercomputing*, p. 131–140, Cambridge, MA, USA, juin 2005.
4. EDLER (J.), *Practical Structures for Parallel Operating Systems*. Thèse de doctorat, New York University, mai 1995.
5. RADOVIĆ (Z.) et HAGERSTEN (E.), « Efficient synchronization for nonuniform communication architectures », dans *SC02*, Baltimore, Maryland, USA, octobre 2002. IEEE.
6. NIKOLOPOULOS (D. S.), POLYCHRONOPOULOS (E. D.) et PAPATHEODOROU (T. S.), « Efficient runtime thread management for the nano-threads programming model », dans *Second IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming*, vol. 1388, p. 183–194, Orlando, FL, USA, avril 1998.
7. OUSTERHOUT (J. K.), « Scheduling techniques for concurrent systems », dans *Third International Conference on Distributed Computing Systems*, p. 22–30, octobre 1982.
8. FEITELSON (D. G.) et RUDOLPH (L.), « Evaluation of design choices for gang scheduling using distributed hierarchical control », *Parallel and Distributed Computing*, vol. 35, 1996, p. 18–34.
9. NAMYST (R.), *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. Thèse de doctorat, Univ. de Lille 1, janvier 1997.
10. DANJEAN (V.), *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs*. Thèse de doctorat, École Normale Supérieure de Lyon, décembre 2004.
11. BARRETO (L. P.) et MULLER (G.), « Bossa : une approche langage à la conception d'ordonnanceurs de processus », dans *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 14)*, Hammamet, Tunisie, avril 2002.
12. STECKERMEIER (M.) et BELLOSA (F.), « Using locality information in userlevel scheduling ». Rapport technique n° TR-95-14, University of Erlangen-Nürnberg – Computer Science Department – Operating Systems – IMMD IV, Martensstraße 1, 91058 Erlangen, Germany, décembre 1995. `http://www4.informatik.uni-erlangen.de/Projects/FORTWIHR/ELiTE/`.
13. SUN microsystems, *Solaris Memory Placement Optimization (MPO)*. `http://iforce.sun.com/protected/solaris10/adoptionkit/tech/mpo/mpo_man.html`.
14. Compaq, *NUMA Scheduling Groups (NSG)*. `http://modman.unixdev.net/?sektion=4&page=numa_scheduling_groups&manpath=OSF1-V5.1-alpha`.